WHITE PAPER

Connecting Relational Databases to Elasticsearch

Use Elasticsearch to add visualization and full text search to your SQL data

Table of Contents

ObjectRocket

You can easily replicate data from a relational database like MySQL or PostgreSQL into Elasticsearch to boost search capabilities or for data analytics purposes. Though NoSQL and Big Data technologies pop up in the news more often with a lot more buzz, relational databases are still alive and well. Almost every customer ObjectRocket works with has some relational data as part of their app, and we occasionally get the question of how best to move or replicate data from these databases. Elasticsearch speeds up and improves search and provides data analytics and visualization when combined with Kibana.

Introduction

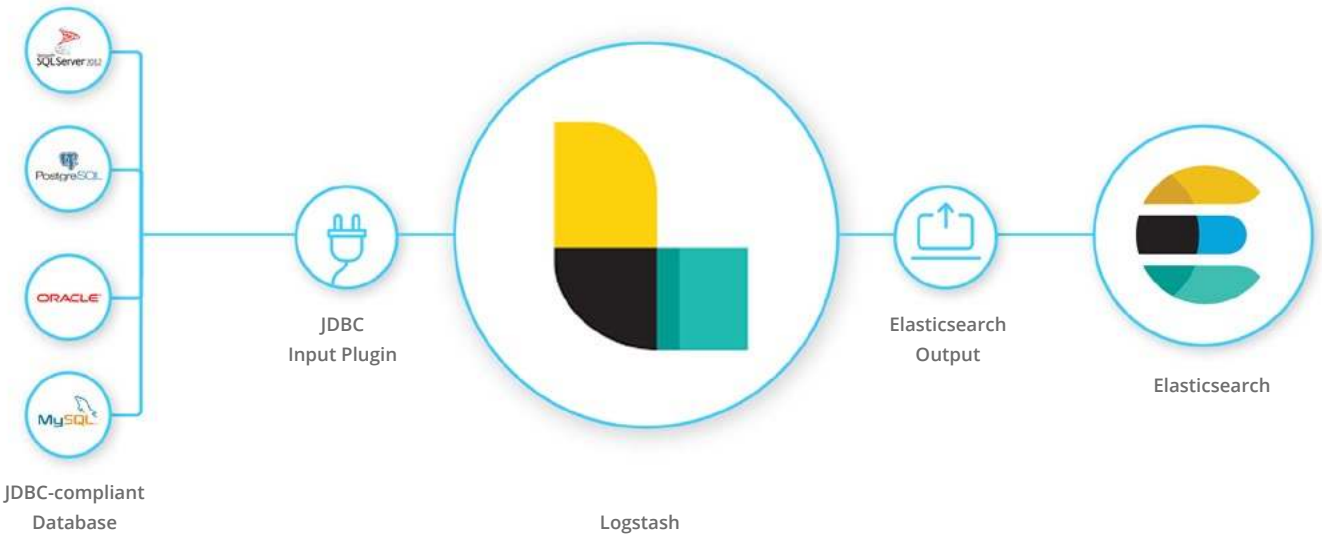
There are a number of ways to connect these two technologies, from writing your own utilities in the language of your choice to off-the-shelf open source tools. In particular, the Elastic Stack provides a number of options in and of itself.

Our Preferred Solution

Among several options, we prefer Logstash with the JDBC input plugin. Here's why:

- Logstash integrates seamlessly and with minimal manual intevention with Elasticsearch.
- The JDBC input plugin only requires MySQL client access to work; some methods of replication require binlogs, which aren't always available in the cloud.
- Using a SQL query to define what to sync is relatively straightforward.
- The filters available in Logstash are extremely powerful and can help flatten out relational data.

An example of the rough architecture using Logstash with the JDBC input plugin:



In this white paper, we'll walk through a specific example. However, the concepts are flexible enough that you can apply them with other technologies. For the rest of this whitepaper, we assume the following:

- You have an Elasticsearch cluster running (example uses version 6.2.4).
  - » If you don't already have an Elasticsearch cluster, give ObjectRocket a try.
- You have a JDBC compatible database running (example uses MySQL 5.7).
  - » You'll want some data in your database and a user that can access the database. For testing, we used the MySQL sample employee dataset.
  - » You can use pretty much any database that has a JDBC driver available.
- You have a compatible JDBC driver for your database (example uses the official MySQL driver).
  - » Whether you're using PostgreSQL, MS SQL Server, Oracle, or others, there is a good chance you can find a supported JDBC driver.
  - » Some NoSQL databases, like MongoDB, even have commercial JDBC drivers available for a fee.
  - » Oracle maintains a list of JDBC compliant vendors.
- You have access to a system for running Logstash. (Our example uses Logstash 6.2.3.)
  - » You can run this on your local machine, with Docker, on a cloud server, or wherever you have some compute available.
  - » The official Logstash guide has good setup instructions for this.
  - » Ensure that both your Elasticsearch cluster and SQL database are reachable from wherever you run Logstash.

Configuring the Datastores and JDBC Driver

The configuration on both the Elasticsearch cluster and on the SQL database are minimal.

In our example, we're running Elasticsearch on the ObjectRocket service. So, we'll create an admin user and whitelist the IP for the Logstash server in the ObjectRocket UI.

On the Elasticsearch side, it's as simple as making sure that you have appropriate access to create an index (or indexes) to copy the data. You can create an index template if you'd like, to preset index settings or some initial mapping, but it's not necessary and you'll probably want to iterate later. So, for this example, we'll just let Elasticsearch auto-generate an index for the first pass.

On the source database side, the setup is similar. You need to make sure you have a user that can access the database(s) you'd like to replicate. MySQL's JDBC driver, which we're using, places almost no requirements on the source database settings, but PostgreSQL, for example, requires some specific settings to ensure JDBC can connect. Therefore, all that is required in our example is that we create a user and grant them access to select the data we want to replicate from the Logstash host.

The JDBC driver itself also requires minimal install and configuration. In the case of the MySQL JDBC driver, setup entails downloading the driver and extracting the appropriate JAR file to a directory that Logstash can get to and ensuring Logstash has the right permissions to access that file. From there, you either need to set the CLASSPATH to include the directory where the driver is, or you can just point to it directly from the Logstash configuration (which we'll show later).

Setting up Logstash Input

Now we just need to tie everything together. First, let's start by setting up the JDBC input plugin and outputting to a local file to test. Here's the initial Logstash configuration file:

```
input {
  jdbc {
    jdbc_driver_library => "/opt/jdbc/mysql-connector-java-5.1.46-bin.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://mysqlserveraddress:3306/employees"
    jdbc_user => "mymysqluser"
    jdbc_password => "notreallyapassword"
    statement => "SELECT * FROM employees LIMIT 10"
    lowercase_column_names => true
  }
}
# filter {
#
# }
output {
  file {
    path => "/tmp/test.log"
    file_mode => 0644
    codec => "json_lines"
  }
}
```

Everything in the example above should look pretty straightforward, but there are a couple of items to note:

- **Jdbc_driver_library:** This is just the name of the JDBC driver file. You can set up the Java CLASSPATH variable to include the location of that file, or you can just use the full path to the jar here. In the example, we dropped ours in a directory we created called /opt/jdbc.
- **Jdbc_driver_class:** This is just the driver class name for the driver you're using. Consult the documentation for your driver.
- **Jdbc_connection_string:** The **jdbc:mysql://** will depend on exactly which type of database you're using, but for MySQL, it should look like **jdbc:mysql://hostnameorIP:port/database**.
- **Statement:** This is just a standard SQL statement to grab whatever data you want from your source. We kept it simple for the first test and grabbed all columns and 10 rows from the employees table.
- **File output plugin:** The file output plugin places the output in a file specified by the path setting. (The example is named "test.log" in the /tmp directory.) Just make sure Logstash has the ability to write to that file and/or create the file if it doesn't exist in that directory.

Since we installed Logstash from the deb package, we just needed to drop the configuration above in a file named **something.conf** in **/etc/logstash/conf.d/**. The default behavior of Logstash 6.x when installed from the deb package is to create a pipelines.yml file in **/etc/logstash**, which then instructs Logstash to load any .conf files from **/etc/logstash/conf.d**. However, you may need to load differently depending on how you installed Logstash. Consult the documentation for your version.

Let's do a test run:

Since I used the debian package on an Ubuntu 16.04 system, **systemd** was used to start and stop Logstash. To run Logstash, I'd use **sudo systemctl start logstash.service**, tail the logs in **/var/log/logstash** to watch status, then stop Logstash with **sudo systemctl stop logstash.service**. This is the general process used to start and stop Logstash in these examples, but consult the Logstash docs for more detail on how to run Logstash if you have a different environment.

Now we check our output file, **/tmp/test.log**, from the example configuration above:


```
{"gender":"F","@timestamp":"2018-04-05T16:15:11.503Z","first_name":"Duangkaew","last_name":"Piveteau","emp_no":10010,"@version":"1","birth_date":"1963-06-01T00:00:00.000Z","hire_date":"1989-08-24T00:00:00.000Z"}
{"gender":"M","@timestamp":"2018-04-05T16:15:11.474Z","first_name":"Georgi","last_name":"Facello","emp_no":10001,"@version":"1","birth_date":"1953-09-02T00:00:00.000Z","hire_date":"1986-06-26T00:00:00.000Z"}
{"gender":"F","@timestamp":"2018-04-05T16:15:11.476Z","first_name":"Bezalel","last_name":"Simmel","emp_no":10002,"@version":"1","birth_date":"1964-06-02T00:00:00.000Z","hire_date":"1985-11-21T00:00:00.000Z"}
{"gender":"M","@timestamp":"2018-04-05T16:15:11.477Z","first_name":"Parto","last_name":"Bamford","emp_no":10003,"@version":"1","birth_date":"1959-12-03T00:00:00.000Z","hire_date":"1986-08-28T00:00:00.000Z"}
{"gender":"M","@timestamp":"2018-04-05T16:15:11.483Z","first_name":"Chirstian","last_name":"Koblick","emp_no":10004,"@version":"1","birth_date":"1954-05-01T00:00:00.000Z","hire_date":"1986-12-01T00:00:00.000Z"}
{"gender":"M","@timestamp":"2018-04-05T16:15:11.484Z","first_name":"Kyoichi","last_name":"Maliniak","emp_no":10005,"@version":"1","birth_date":"1955-01-21T00:00:00.000Z","hire_date":"1989-09-12T00:00:00.000Z"}
{"gender":"F","@timestamp":"2018-04-05T16:15:11.490Z","first_name":"Anneke","last_name":"Preusig","emp_no":10006,"@version":"1","birth_date":"1953-04-20T00:00:00.000Z","hire_date":"1989-06-02T00:00:00.000Z"}
{"gender":"F","@timestamp":"2018-04-05T16:15:11.491Z","first_name":"Tzvetan","last_name":"Zielinski","emp_no":10007,"@version":"1","birth_date":"1957-05-23T00:00:00.000Z","hire_date":"1989-02-10T00:00:00.000Z"}
{"gender":"M","@timestamp":"2018-04-05T16:15:11.500Z","first_name":"Saniya","last_name":"Kalloufi","emp_no":10008,"@version":"1","birth_date":"1958-02-19T00:00:00.000Z","hire_date":"1994-09-15T00:00:00.000Z"}
```

This looks good. We now have consistent json-ified row data. The field names look reasonable, and we don't have any difficult mapping issues, as Elasticsearch should be able to identify all of those data fields automatically.

However, if that didn't work, here's how to troubleshoot:

- Check the Logstash output or logs for any errors. In our example, these files are placed in **/var/log/logstash**. Consult your Logstash docs to be sure to find where they are stored for your use. I
- Confirm you can log into mysql and perform the jdbc statement/query from wherever you're running Logstash using the credentials specified in the JDBC input section of the Logstash configuration.
- Make sure whatever user is running Logstash has the correct permissions to access the JDBC driver jar—whether loaded from the full path, as in the example above, or via the CLASSPATH.
- Confirm the path provided for the jar file and/or make sure the CLASSPATH is set correctly for the user that's actually running Logstash (if not you).
- Confirm that the user running Logstash has access to and the right permissions for wherever you want to place the **test.log** output file.

Setting up the Elasticsearch Output

Now that we know the input side of our filter is working correctly, we need to configure the Elasticsearch end. See the configuration file for the full setup below.

Everything here is, once again, pretty straightforward:

```
input {
  jdbc {
    jdbc_driver_library => "/opt/jdbc/mysql-connector-java-5.1.46-bin.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://mysqlserveraddress:3306/employees"
    jdbc_user => "mymysqluser"
    jdbc_password => "notreallyapassword"
    statement => "SELECT * FROM employees LIMIT 10"
    lowercase_column_names => true
  }
}
# The filter part of this file is commented out to indicate that it is
# optional.
# filter {
#
# }
output {
  elasticsearch {
    id => "esoutput"
    document_id => "%{emp_no}"
    hosts => ["some-es-host", "another-es-host"]
    user => "myelasticsearchuser"
    password => "myelasticsearchpassword"
    ssl => "true"
    index => "mysqlemployees"
  }
}
```

- **Host(s)**: This is a host or list of elasticsearch hosts. If you're using ObjectRocket for Elasticsearch, you can just cut and paste this block from the connection snippets section of the Instance Details screen.
- **User/password**: In the JDBC block, these are your source database credentials. In the Elasticsearch block, these are your Elasticsearch username and password.
- **Index**: If you don't want to use the default of **logstash-%{+YYYY.MM.dd}**, you can specify an index name here.
- **Document_id**: In order to make employees updateable (and some other actions we'll describe later), use the **emp_no** (employee number) field as the Elasticsearch document ID.

Now it's time to rerun Logstash and see what shows up in Elasticsearch. If everything worked, you should see your 10 docs from the test above replicated in Elasticsearch. From the query below, you can see that we have 10 documents in the index and a sample document.

```
GET /mysqlemployees/_search?size=1
```


```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 10,
    "max_score": 1.0,
    "hits": [
      {
        "_index": "mysqlemployees",
        "_type": "doc",
        "_id": "10005",
        "_score": 1.0,
        "_source": {
          "@version": "1",
          "birth_date": "1955-01-21T00:00:00.000Z",
          "@timestamp": "2018-04-20T18:02:02.595Z",
          "gender": "M",
          "first_name": "Kyoichi",
          "emp_no": 10005,
          "hire_date": "1989-09-12T00:00:00.000Z",
          "last_name": "Maliniak"
        }
      }
    ]
  }
}
```


If it didn't work correctly, check your logstash logs and logstash.conf. Also:

- Confirm you can curl the Elasticsearch hosts you specified in the config and get a response with the user and password provided.
- Make sure the user has the right permissions to index documents and create new indexes.

Configuration Wrap-up

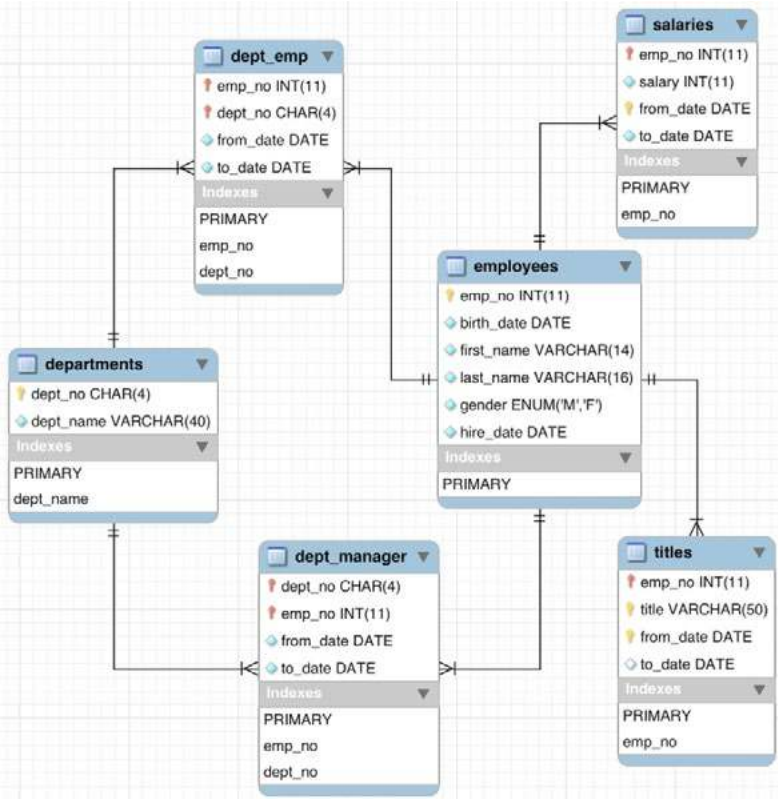
At this point, you should have a repeatable setup for grabbing data from your JDBC connection to your RDMS (i.e. MySQL, PostgreSQL, Oracle) and syncing it with Elasticsearch. Now you can start playing with your SQL query and narrowing it down to the data you actually want. However, there still may be a number of questions. How do you represent multiple relational database tables in Elasticsearch? How do you keep track of updates?

Modeling the Data

There are a number of different ways to model your relational data in Elasticsearch. We'll give you a few examples below so you can decide what is right for your application.

The Sample Data Set

Let's dig into our sample data set to set the stage for how we'll model the data. We used the Employees sample database provided in the MySQL docs, which provides employee records, with title, salary, and department information in additional tables. See the schema below.



As you can see, the **employees** table contains the main employee records, while the **dept_emp**, **dept_manager**, and **titles** tables add additional information about where each employee worked and on which dates.

The big question is how to represent those relations within Elasticsearch. Since Elasticsearch can't really join at query time, how can we make sure we're able to grab all relevant data about an employee with a simple query? There are a few options.

Methods for Modeling the Data

To keep the queries from getting too excessive, we won't worry about salaries and managers for now. For the purposes of this example, we'll just focus on employees, the roles/titles they've had, and the departments they've been in. This requires joining four different tables on the relational side.

Denormalizing Your Data

An easy solution is to just join everything on the SQL side and replicate. You can essentially create an Elasticsearch document for every combination of employee, title, and department they were in.

Here's what the logstash config looks like in this case:

```
input {
  jdbc {
    jdbc_driver_library => "/opt/jdbc/mysql-connector-java-5.1.46-bin.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://mySQLhostname:3306/database"
    jdbc_paging_enabled => true
    jdbc_user => "mysqluser"
    jdbc_password => "mysqlpassword"
    statement => "SELECT e.emp_no as 'employee_number', birth_date, first_name, last_name, gender, hire_date, t.title AS
    'title.name', t.from_date AS 'title.from_date', t.to_date AS 'title. to_date', d.dept_no AS 'department.number',
    ds.dept_name AS 'department.name', d.from_date AS 'department.from_date', d.to_date AS 'department.to_date' FROM
    employees e LEFT JOIN (titles t, dept_emp d, departments ds) ON (e.emp_no = t.emp_no AND e.emp_no = d.emp_no AND
    d.dept_no = ds.dept_no AND t.from_date < d.to_date AND t.to_date > d.from_date)"
    lowercase_column_names => true
  }
}
output {
  elasticsearch {
    id => "esoutput"
    document_id => "%{employee_number}_%{department.number}_%{title.name}_%{title.from_date}"
    hosts => ["eshostone", "eshosttwo"]
    user => "esuser"
    password => "espassword"
    ssl => "true"
    index => "mysqlempdenorm"
  }
}
```

The two key changes are the SQL query, which is now a large set of joins, and the **document_id** on the Elasticsearch side. The SQL query is pretty standard, but the real key is that we're looking for every title an employee had in the company and then making sure we're joining only the department(s) they were in while holding that title.

In the case of the **document_id**, now there can be multiple documents for each employee number since each employee could potentially have had multiple titles/roles within the company. Therefore, we created an ID that takes into account department, title, and start date. The start date was a later add, just to be extra sure that if an employee left a role, then later went back to that same role in that same

department, we won't miss it. None of this is required, though. You can simply let Elasticsearch pick an ID for you. We did it this way so that if we ever wanted to update or overwrite employees, we could base the ID on this identifying information.

One other note is that we've enabled paging with the **jdbc_paging_enabled** setting. As the dataset starts to get larger, you may need to break up the output from the relational database to manage the load. However, this could cause some issues because overlap/reloading parts of the data is possible. What we did with the ID above makes sure we don't create duplicates.

Here's a resulting Elasticsearch doc:

```
{
  "_index": "mysqlempdenorm",
  "_type": "doc",
  "_id": "10007_d008_Staff_1989-02-10T00:00:00.000Z",
  "_score": 1.0,
  "_source":
  {
    "department.from_date": "1989-02-10T00:00:00.000Z",
    "birth_date": "1957-05-23T00:00:00.000Z",
    "@timestamp": "2018-04-10T19:31:38.773Z",
    "title.name": "Staff",
    "gender": "F",
    "hire_date": "1989-02-10T00:00:00.000Z",
    "department.number": "d008",
    "department.name": "Research",
    "department.to_date": "9999-01-01T00:00:00.000Z",
    "title.to_date": "1996-02-11T00:00:00.000Z",
    "employee_number": 10007,
    "first_name": "Tzvetan",
    "last_name": "Zielinski",
    "title.from_date": "1989-02-10T00:00:00.000Z",
    "@version": "1"
  }
}
```

Pros	Cons
• It's pretty easy to identify a specific employee and their role at any point in time. • You get great Kibana support, since you're not using some of the Elasticsearch joins (nested types, parent/child) that aren't well supported in Kibana.	• Potentially larger space usage, since you have every combination of every table. This could be a problem for larger data sets. • It's a pretty expensive query on the SQL side and could cause some performance issues. • Queries that rely on distinct employee counts can be trickier (see below).

Create Arrays or Nested Objects for Departments and Titles

Another option is to just aggregate the titles and departments into arrays/nested objects within the employee docs. This still requires some joining on the SQL side, but also uses the "aggregate" filter within Logstash to combine things like the titles and departments. What we'll end up with is a document per employee with arrays for the roles and departments they've been in. Let's look at how this looks in our Logstash config (example only shows the fields that have changed).

Note 1: The Logstash docs call it out, but know that when using the "aggregate" filter the way we do in the example, it's important to set the number of pipeline workers in Logstash to 1. The filter below requires that all docs with the same **employee_number** are next to each other, which is not guaranteed if you use multiple worker threads. You should also turn off **jdbc_paging_enabled**, which can also cause issues with this filter.

Note 2: You can also do some of this aggregating on the relational side. MySQL, for example, has some functions in 5.7.22 and later that will allow you to combine multiple rows into a JSON array.

```
input {
  jdbc {
    statement => "SELECT e.emp_no as 'employee_number', birth_date, first_name, last_name, gender, hire_date, t.title
    AS 'title.name', t.from_date AS 'title.from_date', t.to_date AS 'title.to_date', d.dept_no AS 'department.
    number', ds.dept_name AS 'department.name', d.from_date AS 'department.from_date', d.to_date AS 'department.
    to_date' FROM employees e LEFT JOIN (titles t, dept_emp d, departments ds) ON (e.emp_no = t.emp_no AND e.emp_no
    = d.emp_no AND d.dept_no = ds.dept_no AND t.from_date < d.to_date AND t.to_date > d.from_date) ORDER BY e.emp_no
    ASC"
  }
}
filter {
  aggregate {
    task_id => "%{employee_number}"
    code => "
      map['employee_number'] = event.get('employee_number')
      map['birth_date'] = event.get('birth_date')
      map['first_name'] = event.get('first_name')
      map['last_name'] = event.get('last_name')
      map['gender'] = event.get('gender')
      map['hire_date'] = event.get('hire_date')
      map['roles'] ||= []
      map['roles'] << {'title.name' => event.get('title.name'),'title.from_date' => event.get('title.from_
      date'),'title.to_date' => event.get('title.to_date'),'department.number' => event.get('department.
      number'),'department.name' => event.get('department.name'),'department.from_date' => event.get('department.
      from_date'),'department.to_date' => event.get('department.to_date')}
      event.cancel()
    "
    push_previous_map_as_event => true
    timeout => 30
  }
}
output {
  elasticsearch {
    document_id => "%{employee_number}"
    index => "mysqlempnested"
  }
}
```

The SQL query is almost exactly the same as before, but the big difference is that we're now specifically ordering by **employee_number**. This is extremely important for the aggregate filter mentioned below because it depends on documents with the same **employee_number** being next to each other.

The aggregate filter that we use will create a temporary map that appends each different title/department that comes through to an array. When it sees a document with a different **task_id/employee number**, it will push the map and its array of roles as a new event (**push_previous_map_as_event => true**). This is very similar to an example in the Logstash documentation.

Outside the Logstash config, you also have a choice of whether you want to put the roles into a nested field or not. If you go the nested route, queries about titles and departments should be more accurate, but the downside is that queries will need to change a bit, and Kibana support for nested fields is not great.

To make these items a nested field, you will need to specify that in the template or mapping before you load the data:

```
PUT /mysqlempnested
{
  “mappings”: {
    “doc”: {
       “properties”: {
          “roles”: {
             “type”: “nested”
          }
       }
    }
  }
}
```

Here's a resulting Elasticsearch doc:


```
{
  “_index”: “mysqlempnested”,
  “_type”: “doc”,
  “_id”: “10007”,
  “_score”: 1.0,
  “_source”:
   {
      “last_name”: “Zielinski”,
      “employee_number”: 10007,
      “first_name”: “Tzvetan”,
      “hire_date”: “1989-02-10T00:00:00.000Z”,
      “@timestamp”: “2018-04-12T17:34:53.300Z”,
      “gender”: “F”,
      “@version”: “1”,
      “birth_date”: “1957-05-23T00:00:00.000Z”,
      “roles”: [
          {
              “title.from_date”: “1996-02-11T00:00:00.000Z”,
              “department.to_date”: “9999-01-01T00:00:00.000Z”,
              “title.name”: “Senior Staff”,
              “department.from_date”: “1989-02-10T00:00:00.000Z”,
              “title.to_date”: “9999-01-01T00:00:00.000Z”,
              “department.name”: “Research”,
              “department.number”: “d008”
          },
          {
              “title.from_date”: “1989-02-10T00:00:00.000Z”,
              “department.to_date”: “9999-01-01T00:00:00.000Z”,
              “title.name”: “Staff”,
              “department.from_date”: “1989-02-10T00:00:00.000Z”,
              “title.to_date”: “1996-02-11T00:00:00.000Z”,
              “department.name”: “Research”,
              “department.number”: “d008”
          }
      ]
   }
}
```

Pros	Cons
• A single doc per employee is easier to manage and may work better with some aggregations.	• It's still a pretty expensive query on the SQL side and could cause some performance issues. • If you go the nested route, there may be some difficulties in Kibana, since nested fields are not supported.

Elasticsearch parent-child index mapping:

```
PUT /mysqlempparentchild
{
  "mappings": {
    "doc": {
        "properties": {
            "doctype": {
                "type": "join",
                "relations": {
                   "employee": "role"
                }
            }
        }
    }
  }
}
```

Use Parent-Child Relationships

Yet another option is to use the parent/child facilities in Elasticsearch. This requires a little more complicated Logstash configuration, since you'll need one query for the parents and one for the children. Logstash 6.x makes this easy, because it includes the ability to create multiple pipelines, so you can just create a configuration file for each type. This is still possible in earlier versions of Logstash, but you'll have to use multiple input plugins with different queries and then use a conditional on the output plugin to determine whether you're loading a parent or child doc. The example will show the latter, since it will work in most versions of Logstash.

One other note is that parent/child has changed significantly in Elasticsearch 6.x because there are no longer multiple mapping types per index. It also seems like the support for parent/child in Logstash 6.x is not quite straightforward when connecting to an Elasticsearch 6.x cluster. The pipeline on page 16 gets it working.

The first order of business is to set up the mapping with the new "join" field type to facilitate the parent/child mapping. Due to the changes in Elasticsearch 6.x, you'll need to create a field of a new "join" type (named "doctype" on the left) and specify the relationships between the various values for that field (in the "relations" sub-field). In our case, our parents will set the "doctype" to "employee" and the children will set it to "role."

Logstash config

```
input {
  jdbc {
    statement => “SELECT emp_no as ‘employee_number’, birth_date, first_name, last_name, gender, hire_date FROM employees e ORDER BY employee_number ASC”
    add_field => { “doctype” => “employee” }
  }
  jdbc {
    statement => “SELECT t.emp_no as ‘employee_number’, t.title AS ‘title.name’, t.from_date AS ‘title.from_date’, t.to_date AS ‘title.to_date’, d.dept_no AS ‘department.number’, ds.dept_name AS ‘department.name’, d.from_date AS ‘department.from_date’, d.to_date AS ‘department.to_date’ FROM titles t LEFT JOIN (dept_emp d, departments ds) ON (t.emp_no = d.emp_no AND d.dept_no = ds.dept_no AND ( t.from_date BETWEEN d.from_date AND d.to_date OR d.from_date BETWEEN t.from_date AND t.to_date)) ORDER BY employee_number ASC”
  }
}
filter {
  if [doctype] != “employee” {
    mutate {
        add_field => {
            “[doctype][name]” => “role”
            “[doctype][parent]” => “%{employee_number}”
        }
    }
  }
}
output {
  if [doctype] == “employee” {
    elasticsearch {
        id => “esparentoutput”
        document_id => “%{employee_number}”
        index => “mysqlempparentchild”
    }
  } else {
    elasticsearch {
        id => “eschildoutput”
        document_id => “%{employee_number}_%{department.number}_%{title.name}_%{title.from_date}”
        index => “mysqlempparentchild”
        routing => “%{employee_number}”
    }
  }
}
```

A summary of the key changes from the previous page:

- You now have two input blocks—one that queries the parent docs and one that queries the child docs. The one for the parent docs sets the doctype, so we can identify them as parents downstream and so we can avoid the need for another mutate statement later to add that field.
- We added a mutate filter that adds fields to the child docs so that Elasticsearch can identify them as child documents and what the parent doc is.
- There are now two output blocks, since the **document_ids** will need to be different between children and parents, with an explicit routing statement for the children.

Pros	Cons
• A single doc per employee is easier to manage and may work better with some aggregations. • For data sets that have a lot of children per parent, you can save some space since you're not having to replicate the parent information in every doc. • You can add children or update parents incrementally without having to reset the entire document.	• Parent-child requires you to use special/specific queries, and the support in Kibana is not there.

At this point, let's look at the parent-child mechanics in Elasticsearch 6.x a bit, since they're new. The "join" field can operate slightly differently depending on whether you're sending a parent or child document. For parents, it's just as easy as setting the join field to the parent type. (We do this by setting "doctype" to "employee" in the input block.)

For children, you set this field a little differently. You need to set a "name" sub-field to the type of document ("role" in our case), and then set a "parent" subfield to the id of the parent doc ("**${employee_number}**" in our case). Finally, the child documents need to have their routing set to the parent ID (unless you're using something else to route your parent docs) to ensure they end up on the same shard.

Now we have parent/child mapped Elasticsearch documents:


```
{
  “_index”: “mysqlempparentchild”,
  “_type”: “doc”,
  “_id”: “10007”,
  “_score”: 1.0,
  “_source”: {
    “@version”: “1”,
    “employee_number”: 10007,
    “first_name”: “Tzvetan”,
    “birth_date”: “1957-05-23T00:00:00.000Z”,
    “gender”: “F”,
    “last_name”: “Zielinski”,
    “@timestamp”: “2018-04-14T04:06:31.926Z”,
    “hire_date”: “1989-02-10T00:00:00.000Z”,
    “doctype”: “employee”
  }
},
{
  “_index”: “mysqlempparentchild”,
  “_type”: “doc”,
  “_id”: “7qlVwmIBNwBnVopfTOPz”,
  “_score”: 1.0,
  “_routing”: “10007”,
  “_source”: {
    “title.name”: “Senior Staff”,
    “department.to_date”: “9999-01-01T00:00:00.000Z”,
    “@version”: “1”,
    “department.name”: “Research”,
    “title.to_date”: “9999-01-01T00:00:00.000Z”,
    “@timestamp”: “2018-04-14T04:06:32.170Z”,
    “department.number”: “d008”,
    “employee_number”: 10007,
    “title.from_date”: “1996-02-11T00:00:00.000Z”,
    “doctype”: {
        “parent”: “10007”,
        “name”: “role”
    },
  “department.from_date”: “1989-02-10T00:00:00.000Z”
  }
}
```

Query Examples

Since the data will be modeled a little differently, let's look at a couple of queries to see the differences.

Show me employees that are there currently:

```
{
  "sort": {"employee_number": "asc"},
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "title.to_date": {
              "gt": "now"
            }
          }
        },
        {
          "range": {
            "department.to_date": {
              "gt": "now"
            }
          }
        }
      ]
    }
  }
}
```

```
{
  "sort": {"employee_number": "asc"},
  "query": {
    "nested": {
      "path": "roles",
      "query": {
        "bool": {
          "filter": [
            {
              "range": {
                "roles.title.to_date": {
                  "gt": "now"
                }
              }
            },
            {
              "range": {
                "roles.department.to_date": {
                  "gt": "now"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

```
{
  "sort": {"employee_number": "asc"},
  "query": {
    "has_child": {
      "type": "role",
      "inner_hits": {},
      "query": {
        "bool": {
          "filter": [
            {
              "range": {
                "title.to_date": {
                  "gt": "now"
                }
              }
            },
            {
              "range": {
                "department.to_date": {
                  "gt": "now"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

As you can see, the query is VERY similar, with the only real difference being the need for a "nested" query in the nested case and a "has_child" query in the parent-child case.

All three queries return 240,124 hits. The only difference is how they're displayed. Here's an example for each:

```
{
  "_index": "mysqlempdenorm",
  "_type": "doc",
  "_id": "10004_d004_Senior Engineer_1995-12-01T00:00:00.000Z",
  "_score": null,
  "_source": {
    "last_name": "Koblick",
    "birth_date": "1954-05-01T00:00:00.000Z",
    "title.to_date": "9999-01-01T00:00:00.000Z",
    "department.number": "d004",
    "title.name": "Senior Engineer",
    "title.from_date": "1995-12-01T00:00:00.000Z",
    "@timestamp": "2018-04-17T18:13:22.355Z",
    "@version": "1",
    "department.to_date": "9999-01-01T00:00:00.000Z",
    "gender": "M",
    "first_name": "Chirstian",
    "employee_number": 10004,
    "department.from_date": "1986-12-01T00:00:00.000Z",
    "hire_date": "1986-12-01T00:00:00.000Z",
    "department.name": "Production"
  },
  "sort": [
    10004
  ]
},
```


```
{
  "_index": "mysqlempnested",
  "_type": "doc",
  "_id": "10004",
  "_score": null,
  "_source": {
    "employee_number": 10004,
    "hire_date": "1986-12-01T00:00:00.000Z",
    "last_name": "Koblick",
    "@timestamp": "2018-04-17T16:14:48.340Z",
    "roles": [
      {
        "department.name": "Production",
        "title.to_date": "1995-12-01T00:00:00.000Z",
        "department.number": "d004",
        "department.from_date": "1986-12-01T00:00:00.000Z",
        "title.from_date": "1986-12-01T00:00:00.000Z",
        "department.to_date": "9999-01-01T00:00:00.000Z",
        "title.name": "Engineer"
      },
      {
        "department.name": "Production",
        "title.to_date": "9999-01-01T00:00:00.000Z",
        "department.number": "d004",
        "department.from_date": "1986-12-01T00:00:00.000Z",
        "title.from_date": "1995-12-01T00:00:00.000Z",
        "department.to_date": "9999-01-01T00:00:00.000Z",
        "title.name": "Senior Engineer"
      }
    ],
    "birth_date": "1954-05-01T00:00:00.000Z",
    "first_name": "Chirstian",
    "@version": "1",
    "gender": "M"
  },
  "sort": [
    10004
  ]
},
```


```
{
  "_index": "mysqlempparentchild",
  "_type": "doc",
  "_id": "10004",
  "_score": null,
  "_source": {
    "@version": "1",
    "birth_date": "1954-05-01T00:00:00.000Z",
    "employee_number": 10004,
    "first_name": "Chirstian",
    "gender": "M",
    "last_name": "Koblick",
    "@timestamp": "2018-04-17T16:53:49.011Z",
    "hire_date": "1986-12-01T00:00:00.000Z",
    "doctype": "employee"
  },
  "sort": [
    10004
  ],
  "inner_hits": {
    "role": {
      "hits": {
        "total": 1,
        "max_score": 0.0,
        "hits": [
          {
            "_type": "doc",
            "_id": "10004_d004_Senior Engineer_1995-12-01T00:00:00.000Z",
            "_score": 0.0,
            "_routing": "10004",
            "_source": {
              "department.from_date": "1986-12-01T00:00:00.000Z",
              "department.to_date": "9999-01-01T00:00:00.000Z",
              "@timestamp": "2018-04-17T16:53:51.842Z",
              "doctype": {
                "name": "role",
                "parent": "10004"
              },
              "title.name": "Senior Engineer",
              "title.to_date": "9999-01-01T00:00:00.000Z",
              "@version": "1",
              "employee_number": 10004,
              "department.name": "Production",
              "department.number": "d004",
              "title.from_date": "1995-12-01T00:00:00.000Z"
            }
          }
        ]
      }
    }
  }
},
```


The big difference here is the way the roles are displayed. The denormalized data is giving us exactly the role and employee data we want in a single document. The nested query will return the entire document, which will include all roles, so further filtering is required. The parent-child query will simply return the parent employee record. However, you can also provide just the matching role by using the **inner_hits** option in the query.

How many employees there have been (all time):

Though a pretty simple request, this one can get a little tricky on the denormalized data.

```
{
  "aggs": {
    "card": {
      "cardinality": {
        "field": "employee_number",
        "precision_threshold": 40000
      }
    }
  }
}
```

```
{
}
```

```
{
  "query": {
    "match": {"doctype": "employee"}
  }
}
```

We'll start with nested and parent-child first, because those are the easiest. Since nested has just stored all of the titles/roles in an array, you can just do a search and see how many hits you get. Since each employee has a doc, it will give you an accurate count. Parent-child is similarly easy. You just have to look for all documents that have the employee doctype, and the number of hits is your answer.

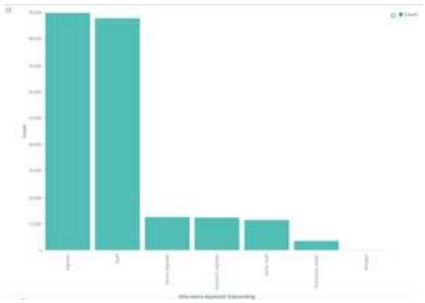
The denormalized data gets a bit trickier. Since each employee can show up in multiple documents, if they've had multiple roles, there's no easy way to pick out unique or distinct employee IDs. There are a few easy approximations, like using the cardinality aggregation, but it bears mentioning that cardinality is not guaranteed to be accurate for high cardinality fields. You can crank up the precision to the max, but it's still an approximation. There are definitely more computationally expensive ways to get the answer, either with some scripting on the Elasticsearch side or via some client-side massaging, but the point is that once the data is denormalized, it can sometimes be hard to extract out data like this.

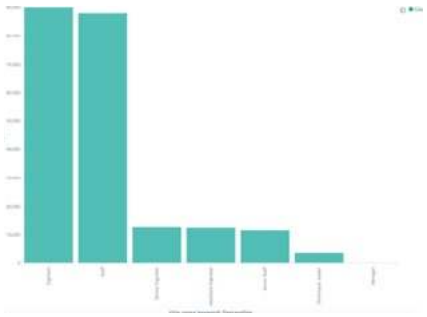
Kibana Examples

Here are a few examples that show the differing levels of Kibana support.

Top 10 Job Titles on January 1, 1990

First, we'll look at a breakdown of the top 10 employee titles.





We got the results we need from both denormalized and parent-child, but nested returned nothing. The reason nested failed is that you need to use a nested query to be able to return fields that are nested. Parent-child, on the other hand, was able to return documents because the child documents are actual queryable documents

However, if we need to see visualizations on both employee and title data, the story changes.

Last Names of People in the Department Development

If we then try to visualize the last names of people in the department development, the results look like this:

Now you can see that only the denormalized data can display what you're looking for. The nested case fails for the same reason as above. Role data is hidden unless you're using the nested query. Parent-child fails this time because you can easily query and aggregate on parent and child documents separately, but the only way to link them is with a parent-child query, which Kibana does not support.

Essentially, Kibana is a mixed bag when it comes to visualizing the different ways of mapping the data.

How to Refresh the Data

Another major aspect of how to model this data is how and how often to load new data. The preceding sections focus on dumping the entire contents of the source database as one large import. However, for some data you may want to only add updated data as it changes. Here's a brief review of some options for how to handle updates.

Daily Snapshots

The easiest solution is to just take a snapshot on a periodic basis (i.e. hourly, daily) of the full dataset, or the part of the data set you're interested in. In this case, you just set your queries like we did above and then use the schedule field in the JDBC plugin to set how often that query runs.

Also, in the examples above, we use a static index name, which basically means we'll always overwrite our data as we update it and only keep a single index for our MySQL data, which will not account for deleted rows unless we clear out the old index first.

An alternative method is to use a dynamic index name by including something unique like a date in the index name, so that each time the pipeline runs (or on some schedule) it drops the results in a new index. This can be helpful if you want regular full snapshots of the data and want to watch how the overall data set changes.

Update as New Rows Appear

The other main option is to track where you left off last time you ran a query and just update incrementally from that point. Logstash includes a number of settings and special fields to help you manage this, like the **sql_last_value** field, **tracking_column setting**, and **use_column_value** setting. The idea is that you can track, for example, the last ID you transferred to Elasticsearch, then only add new records as they come in. There are a few resources online that document this route fairly well.

This can be helpful, can create a smaller data footprint, and can minimize the amount of data transferred between the databases, but there are some downsides. First of all, you need a good column to track that always moves in a consistent direction (like a "last updated" timestamp). Second, this doesn't really handle deletes on the source very well, so it won't create a perfect picture of your relational data.

For example, in the data set above, you can set your SQL queries to only return documents that have a **hire_date** greater than the last time Logstash ran. This will grab all new employees but will miss any deleted employees, employees that have some other attribute changed, and employees that just changed roles. You can work around some of these limitations, but it can get complicated. It really depends on what data is available in MySQL, whether it allows you to identify changes, and exactly when the changes occurred.

How to Choose

So how do you choose? It depends on your data. It boils down to a number of factors, like what kind of data you have, what the schema looks like, how much data you have, and how you want to use that data. It's no surprise that everyone's situation will be different, but here are some guidelines based on two of the largest factors: data size and use case.

The general logic here is that small data sets are inexpensive to process and store, so regular full snapshots are the absolute easiest way to load up the data.

When it comes to visualization or analytics, the ability to see all of your data correctly in Kibana gets a lot of weight, so denormalized or non-nested arrays is where we lean in these cases. For example, we at ObjectRocket use Elasticsearch for analytics/visualization of how our fleet is being used. To do this, we use a denormalized daily index of everything and the history for a set amount of time. However, the big downside with the denormalized data can be aggregating certain types of data or counting things. In our example above, trying to get an all-time employee count on our denormalized data is tricky since each employee can have multiple docs and looking at the cardinality of employee numbers is not guaranteed to be accurate. This is where a non-nested array could help at the expense of some potentially incorrect results in other areas. You may be able to work around this with another index of just employees or metrics you grab with SQL, or something similar.

The same goes on the search side. Though with denormalized data, you'll need to be careful about duplicate responses, the speed and ease of not having to use any specialized queries with denormalized data can outweigh the advantages of a nested field. However, depending on how you want to query the joined data, the nested option may be better and give more accurate results.

Data Set Size \ Use Case	Visualization or Analytics	Search
Small	Denormalized or non-nested arrays with regular full snapshots	Denormalized or nested with regular full updates
Large	Denormalized or non-nested arrays with incremental updates	Parent-child or nested with incremental updates

For data sets that are large enough to be troublesome for your ES cluster, the story becomes a little different. We tend to choose the denormalized route for visualizations and analytics because support is lacking for nested and parent-child in Kibana. However, you may just want to shrink which data you keep or minimize update size by only incrementally updating the data.

On the search side, parent-child offers some nice advantages for incremental updates, like being able to update parents and children separately. It also may shrink your data footprint in some cases, like scenarios where each parent has lots of children. However, parent-child queries can be many times slower than nested and denormalized queries, so the answer you pick here will really have to do with query speed expectations. If speed is a factor, you may want to use the nested case instead or work around the duplicates in the denormalized case.

Closing and Alternatives

You can see that there's quite a bit of flexibility for modeling data in Elasticsearch to match your use case. However, certain things like staying synchronized on updates and deletes are a little problematic in this scenario. Though most have reasonable workarounds, there are also alternatives, like go-mysql-elasticsearch, that are worth considering.

About ObjectRocket

ObjectRocket's technology and expertise helps businesses build better apps, faster so developers can concentrate on creating applications and features without having to worry about managing databases. We'll migrate your data at no cost and with little-to-no downtime. Our DBAs do all the heavy lifting for you so you can focus on your builds. We provide 24x7x365 expert support and architecture services for MongoDB, Elasticsearch, Redis, and Hadoop instances in data centers across the globe.

If you want to connect your database with Elasticsearch, remember that ObjectRocket can help you through it. We offer fully managed database-as-a-service solutions, and we can free up your developers to focus on building your app by taking the database maintenance piece off your hands.

GET STARTED WITH A CONSULTATION

Date Modified: 05/16/2018